



ТИНЬКОФФ

Многопоточность

Продвинутые темы



О себе



Пришел стажером в 2020 году



Старший разработчик в Тинькофф



Развиваю партнерские сервисы Выгоды



Преподаю в Тинькофф образовании



Программа модуля



Основы работы с потоками и их устройство

Рассмотрим что такое потоки и процессы, когда, как и зачем их использовать



Использование потоков в реальном коде

Разберем что предлагает библиотека Java для работы с потоками



Продвинутые темы

Изучим Java memory model и посмотрим на виртуальные потоки

План лекции



Java memory model

Поговорим про JMM и зачем оно нам нужно



Виртуальные потоки

Настала пора java 21, разберемся что за виртуальные потоки они нам принесли



ТИНЬКОФФ

Java Memory Model

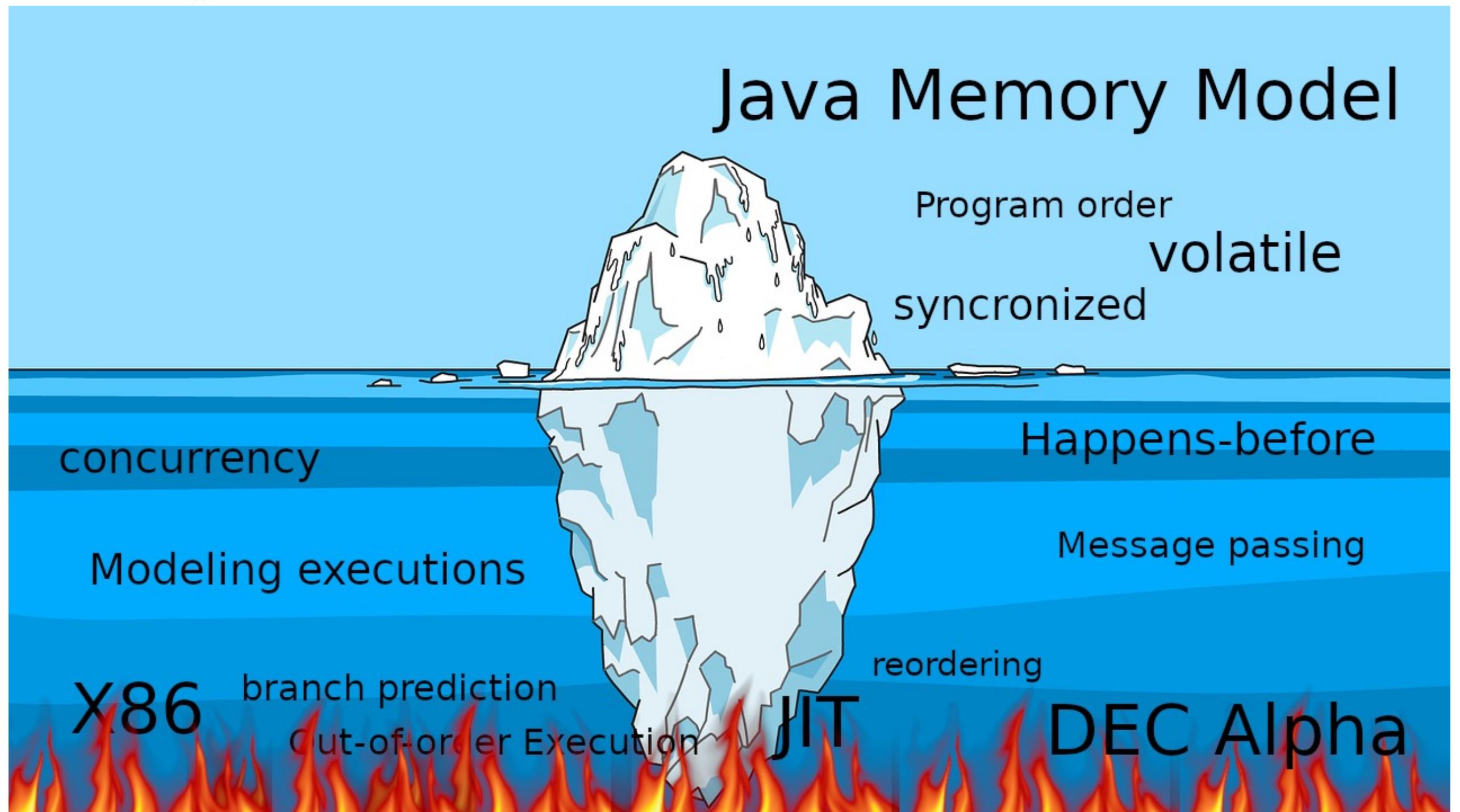
JMM



Что такое JMM?



Зачем нам JMM?



JMM

- В реальной жизни редко используем
- Позволяет спуститься на уровень ниже
- Расширяет наш кругозор в Java
- Полезно на собеседованиях



Numbers every programmer should now

action	approximate time (ns)
typical processor instruction	1
fetch from L1 cache	0.5
branch misprediction	5
fetch from L2 cache	7
mutex lock/unlock	25
fetch from main memory	100
2 kB via 1 GB/s	20.000
seek for new disk location	8.000.000
read 1 MB sequentially from disk	20.000.000

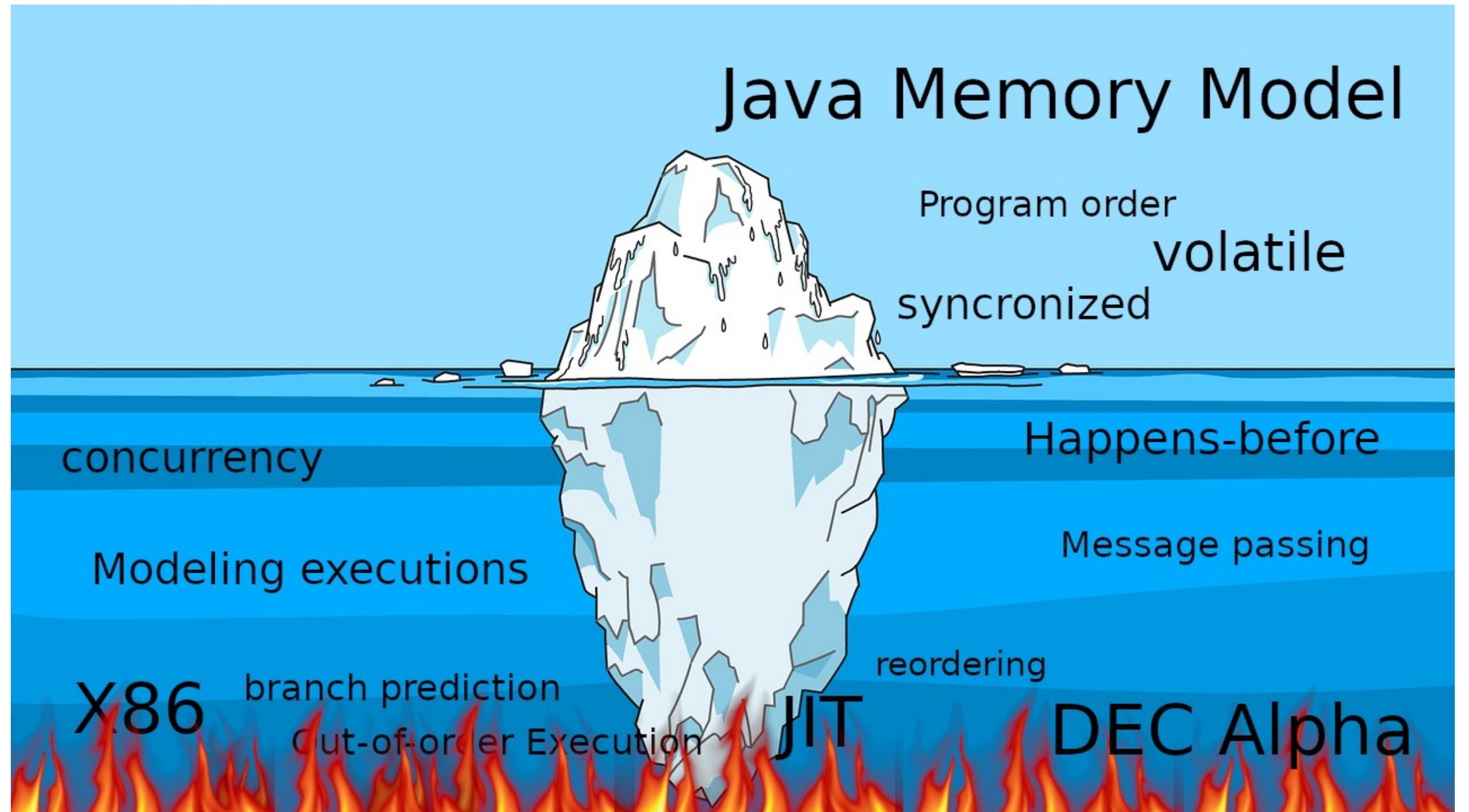
JMM



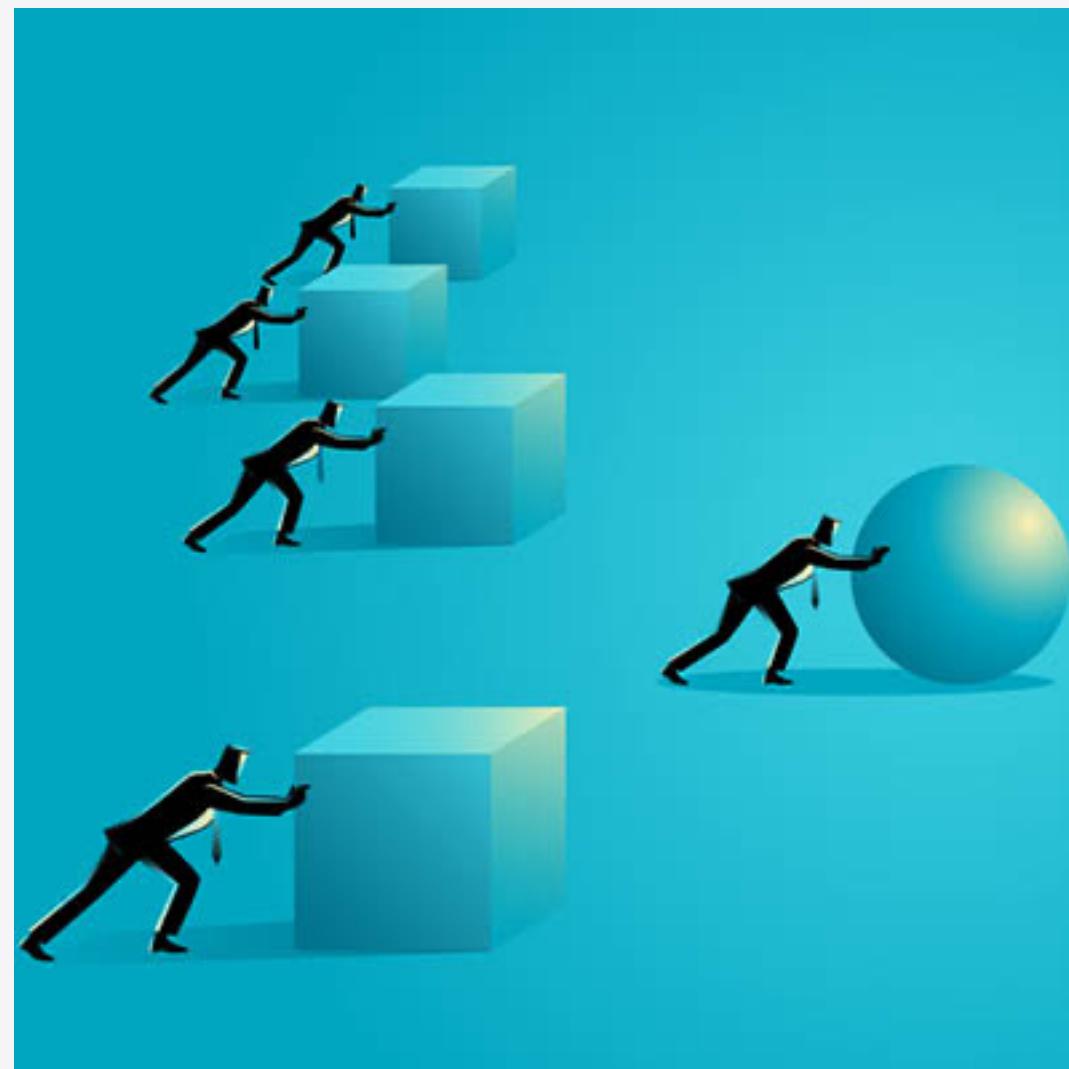
От кода, который мы
написали происходит много
оптимизаций



Оптимизации происходят на
разных уровнях

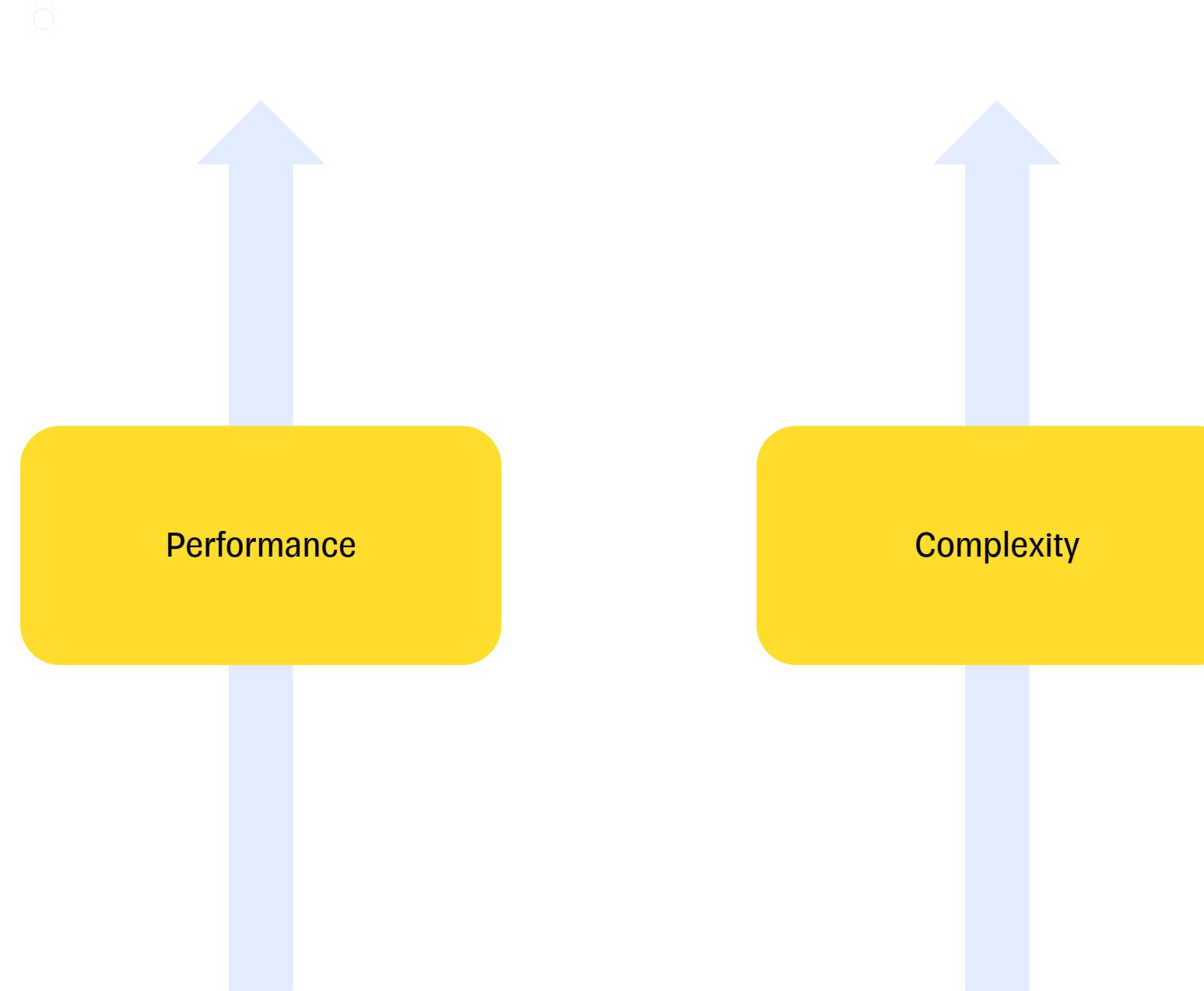


Оптимизации



- Многоядерность
- Out of order execution
- Branch prediction
- Переупорядочивание на уровне компилятора
- Локальный кэш процессора

Оптимизации



Пример 1



Код выполняется в одном потоке



Мы ожидаем понятный результат

The screenshot shows a mobile application interface with a dark-themed code editor. At the top, there are three colored dots: red, yellow, and green. Below them, the following code is displayed:

```
a = 5;  
b = 7;  
int r1 = a; /* always 5 */  
int r2 = b; /* always 7 */
```

Пример 1



Но в реальности исполнение
могло быть другим



Произошел **reordering**



```
b = 7;  
a = 5;  
int r2 = b; /* 7 */  
int r1 = a; /* 5 */
```

Пример 2



Что будет в t1 и t2 ?

Thread 1



```
a = b = 0;
```

```
t1 = b  
a = 1
```

Thread 2



```
a = b = 0;
```

```
t2 = a  
b = 1
```

Пример 2

→ (0, 0)

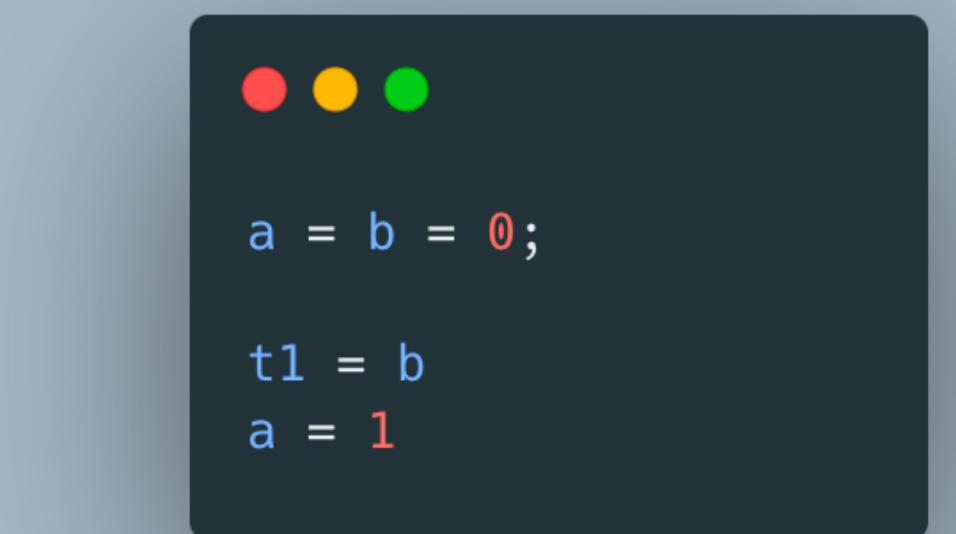
→ (0, 1)

→ (1, 0)

→ (1, 1)

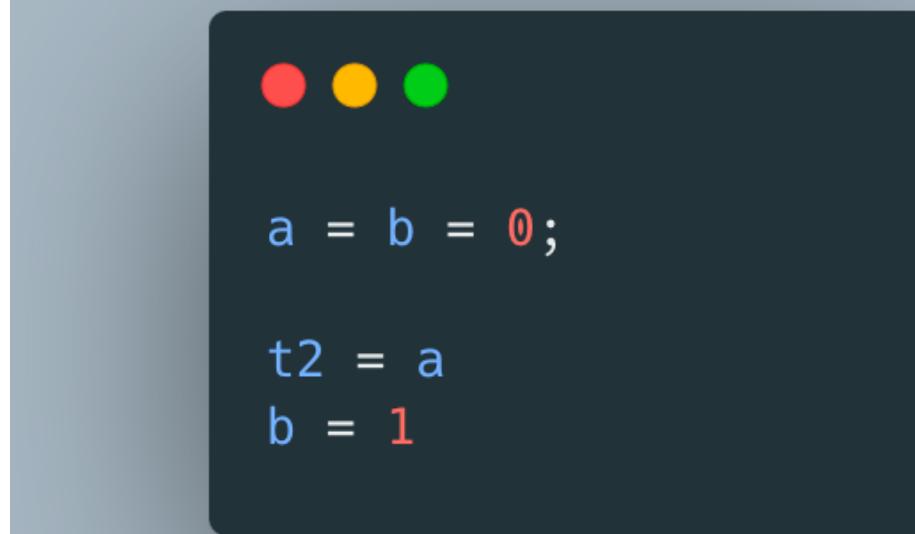
Thread 1

Thread 2



Thread 1 window showing local variables (red, yellow, green dots) and code:

```
a = b = 0;  
t1 = b  
a = 1
```



Thread 2 window showing local variables (red, yellow, green dots) and code:

```
a = b = 0;  
t2 = a  
b = 1
```

Пример 2



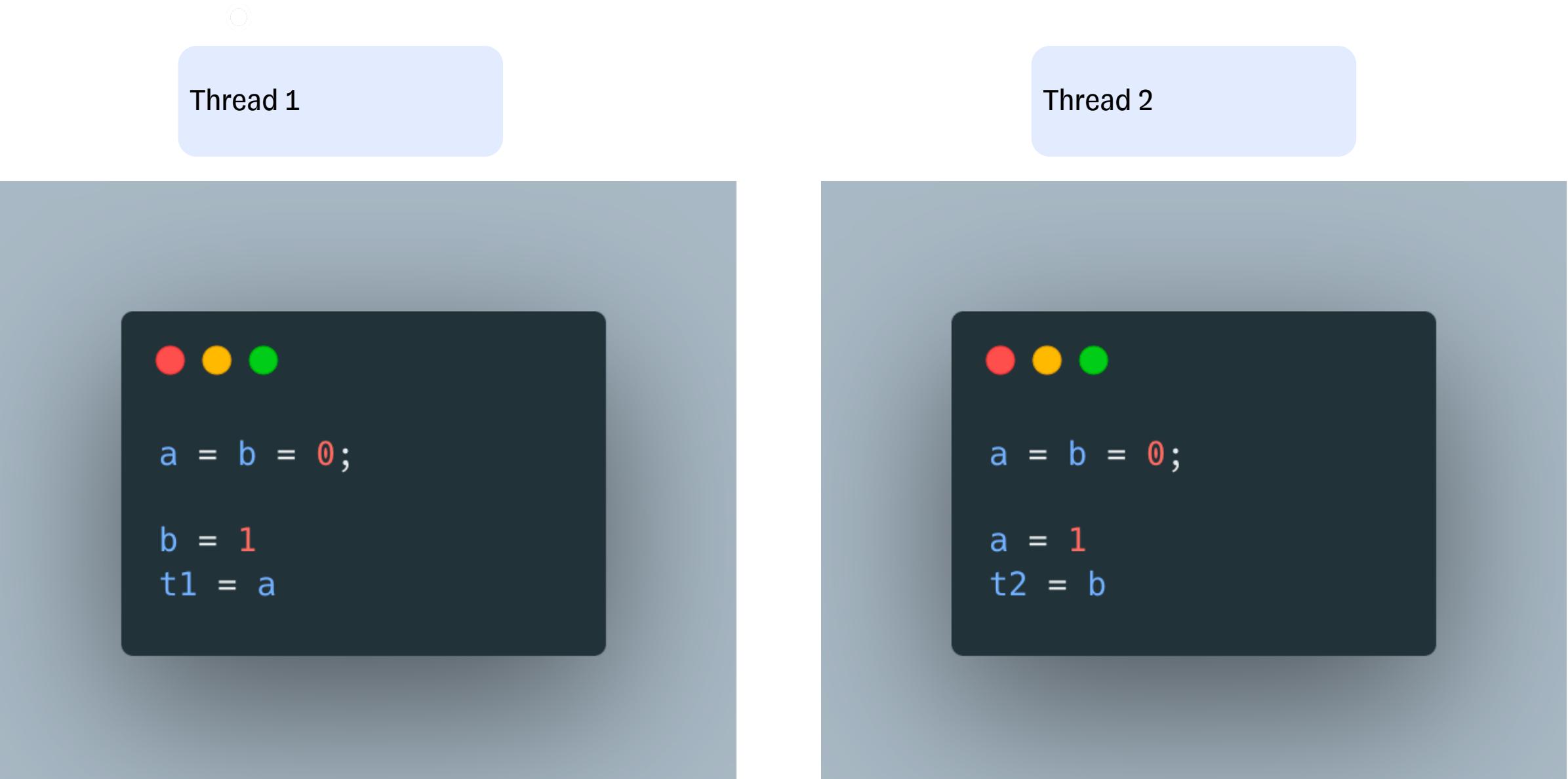
Правильно все



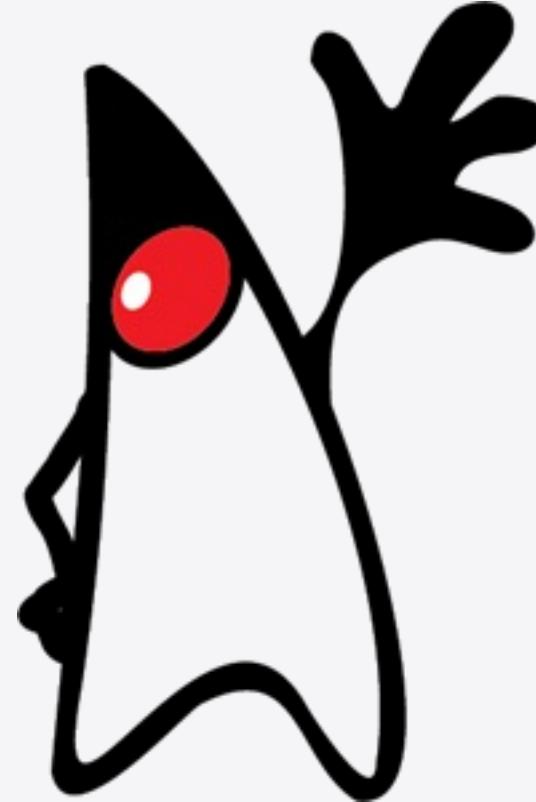
Опять наш **reordering**



Компилятор же не знает на
каких ядрах будет
запускаться код, ему все
равно



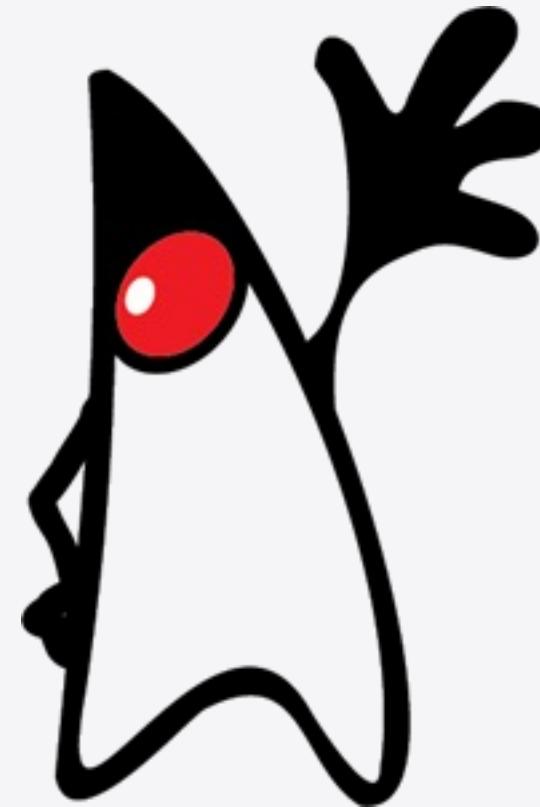
JMM



- - Это спецификация, которая отвечает на вопрос, что мы можем получить при чтении из переменных
 - В основном нам полезна при многопоточном исполнении
 - JMM описывает какое исполнение программы является валидным



Это спецификация, которая отвечает на вопрос, что мы можем получить при чтении из переменных



```
class SingleThreaded {  
  
    int foo = 0;  
  
    void method() {  
        foo = 1;  
        assert foo == 1;  
    }  
}
```

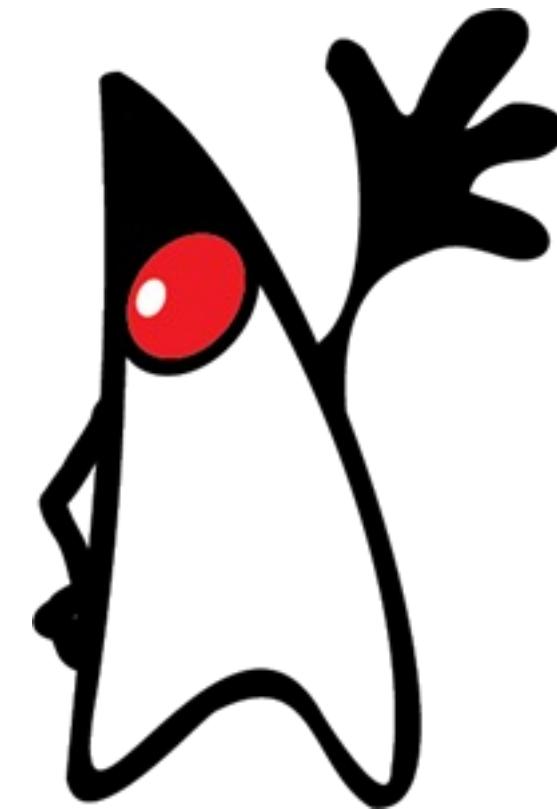
write action
read action



program order

JMM

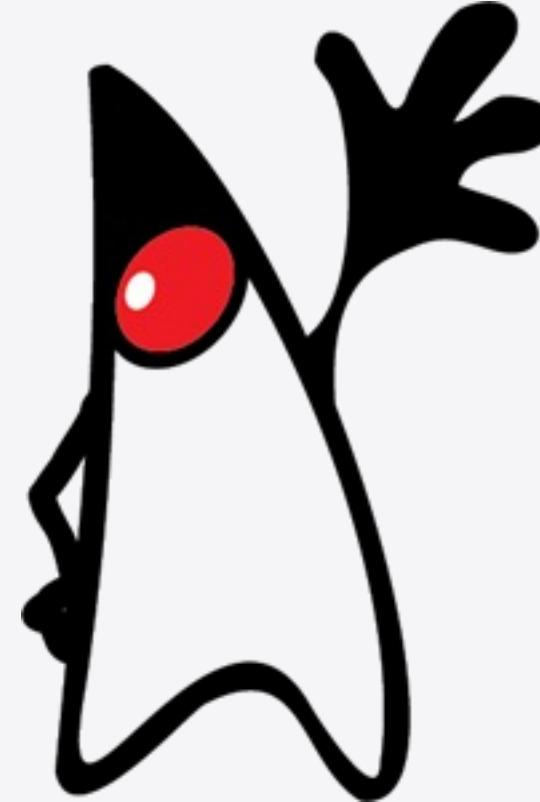
- Program order – как написано в коде
- Execution order – как исполняется на процессоре
- Visibility order – как изменения одного потока будут видны в другом



Program order

Execution order

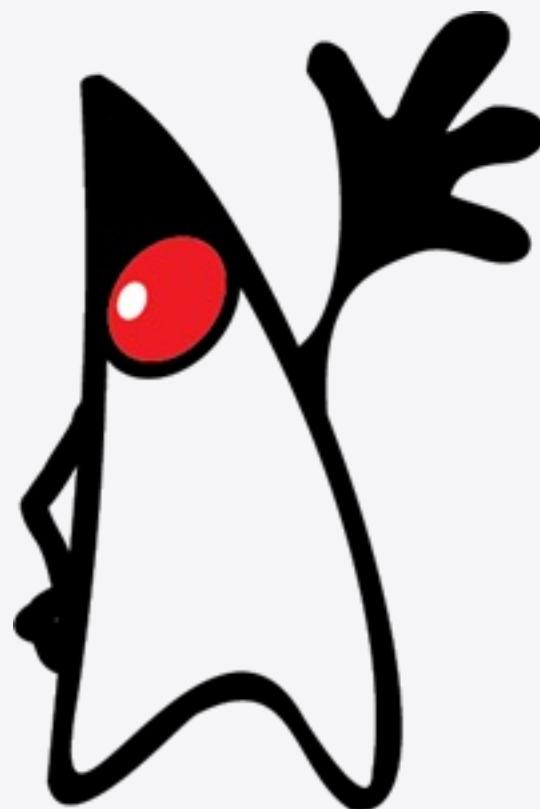
Visibility order



- ➡ Разработчики знают поведение своей программы
- ➡ Нужно знание это передать среде исполнения
- ➡ Модель памяти = **trade-off** между
 - ➡ Сложностью программирования на языке
 - ➡ Сложностью реализации языка
 - ➡ Сложностью **hardware**

JMM

Строительные блоки



final

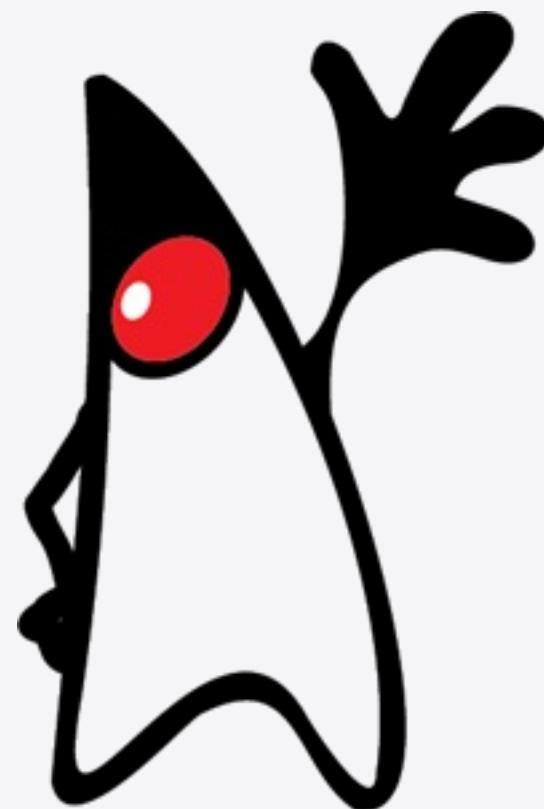
Synchronized

volatile

Lock

JMM

Строительные блоки



final

Synchronized

volatile

Lock

Используя эти **keywords** мы можем сказать JVM, как
правильно выполнить код

Но для этого необходимо понимать некоторые правила игры

Happens before



JMM определяет частичный порядок, который называется **happens-before**



Happens-before – транзитивное отношение

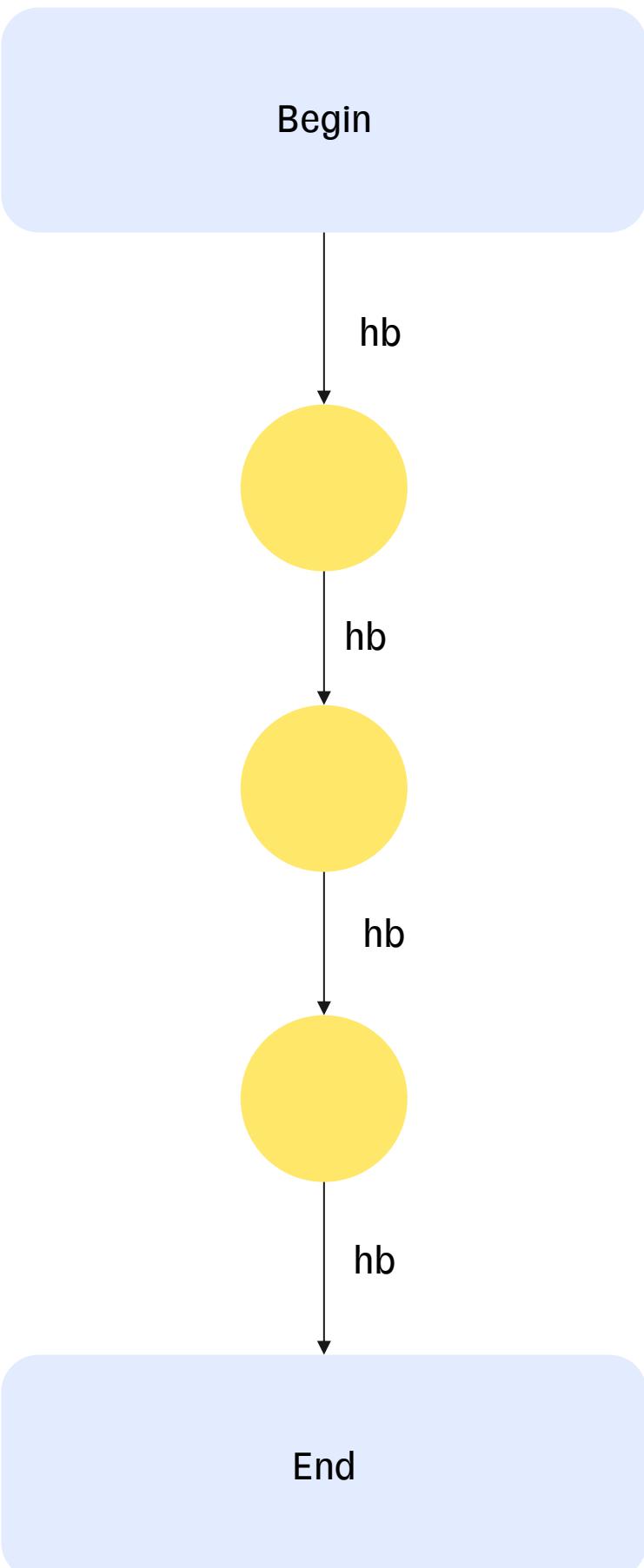


Чтобы действие В видело изменение действия А, достаточно, чтобы А *hb* В

Program order rules

→ В однопоточных программах все действия happens-before в порядке их определения

→ Happens-before – транзитивное отношение



Thread start & thread termination rule

→ Вызов `thread.start()` happens-before всех операций в потоке `thread`

→ Любая операция в потоке `threadA` happens-before обнаружения другим потоком завершения `threadA. threadA.join()`



```
public static void main(String[] args) {
    Thread thread = new Thread(() -> {
        System.out.println("Hello from a thread!");
        System.out.println("I'm happens before the thread.join()!");
    });
    thread.start();
}
```

The screenshot shows a Java code editor with a dark theme. At the top left, there are three colored circular icons: red, yellow, and green. The code itself is a simple main method. It creates a new thread using a lambda expression and starts it. Inside the thread's run block, two println statements are present: one that prints "Hello from a thread!" and another that prints "I'm happens before the thread.join()!". The code is syntax-highlighted, with 'public' in purple, 'static' in blue, 'void' in orange, 'main' in light blue, 'String' in pink, 'args' in light blue, 'Thread' in blue, 'System.out.println' in light blue, and the strings in green.

Volatile

Запись в volatile-
переменную happens-before
чтения из этой переменной в
другом потоке



```
static volatile int nextCustomerNumber = 0;

public static void main(String[] args) throws Exception {
    new CustomerController(4).start();
    new Queue().start();
}

static class CustomerController extends Thread {
    private final int customerNumber;
    public CustomerController(int customerNumber) {
        this.customerNumber = customerNumber;
    }
    @Override
    public void run() {
        while (nextCustomerNumber < customerNumber) {
        }
        System.out.printf("Клиент наконец дошел %d\n", nextCustomerNumber);
    }
}

static class Queue extends Thread {
    @Override
    public void run() {
        while (nextCustomerNumber < 20) {
            System.out.printf("Вызываем клиента #%d\n", nextCustomerNumber++);
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Барьер



Что здесь может
переставиться?

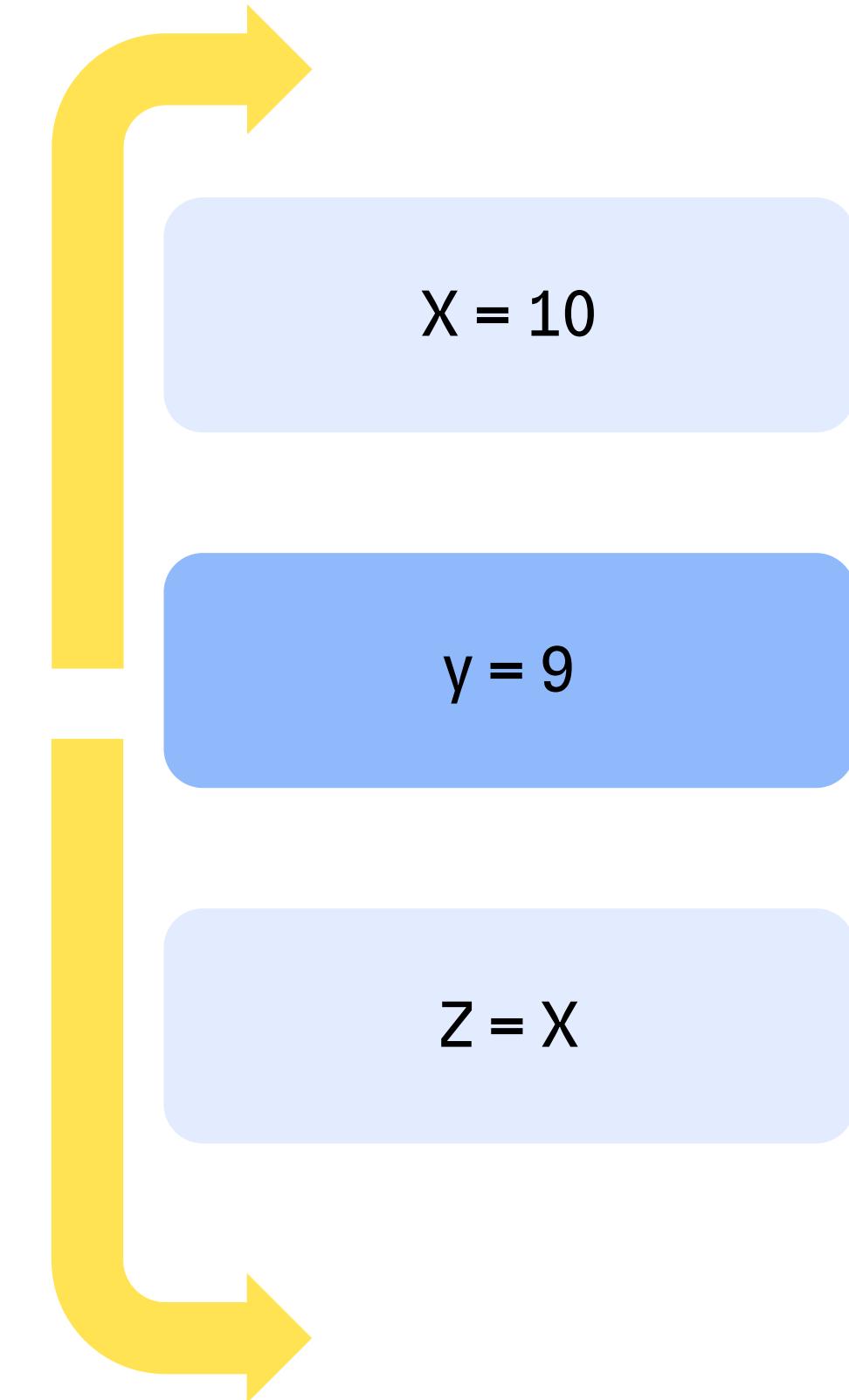
$X = 10$

$y = 9$

$Z = X$

Барьер

- ➡ **$Y = 9$ может переставляться**
- ➡ **$X = 10$ и $Z = X$ связаны
зависимостью по данным**
- ➡ **$X = 10$ и $Z = X$ связаны
зависимостью по данным**



Барьер



Операции с переменной `y` –
становятся как барьер



Операции выше и ниже не
могут пройти сквозь него

X = 10

y = 9 // y - volatile

A = 10

Z = 15

B = y // y - volatile

C = 15

Барьер



Операции с переменной y –
становятся как барьер



Операции выше и ниже не
могут пройти сквозь него



Какие мы можем сделать
выводы?

X = 10

y = 9 // y - volatile

A = 10

Z = 15

B = y // y - volatile

C = 15

Барьер



Операции с переменной y –
становятся как барьер



Операции выше и ниже не
могут пройти сквозь него



$C = 15$ hb $X = 10$

$X = 10$

$y = 9 // y - volatile$

$A = 10$

$Z = 15$

$B = y // y - volatile$

$C = 15$

Final - поля

- Если ссылка на объект не утекает во время выполнения конструктора – final поля доступны всем без синхронизации
- Immutable state удобно использовать при многопоточном исполнении



Final



Поток reader точно увидит в f.x значение 3, так как final



Но значение в f.y может еще быть 0



```
public class FinalFieldExample {  
  
    final int x;  
    int y;  
    static FinalFieldExample f;  
  
    public FinalFieldExample() {  
        x = 3;  
        y = 4;  
    }  
  
    static void writer() {  
        f = new FinalFieldExample();  
    }  
  
    static void reader() {  
        if (f != null) {  
            int i = f.x;  
            int j = f.y;  
            System.out.printf("i = %d, j = %d\n", i, j);  
        }  
    }  
}
```

Final



Поток reader точно увидит в f.x значение 3, так как final



Но значение в f.y может еще быть 0

```
public class FinalFieldExample {

    final int x;
    int y;
    static FinalFieldExample f;

    public FinalFieldExample() {
        x = 3;
        y = 4;
    }

    static void writer() {
        f = <allocate FinalFieldExample>;
        f.<init>();
    }

    static void reader() {
        if (f != null) {
            int i = f.x;
            int j = f.y;
            System.out.printf("i = %d, j = %d\n", i, j);
        }
    }
}
```

JMM Monitor Lock Rule

- Разблокировка (unlocking) happens-before другой блокировки (locking) тоже замка
- Защищенные блокировкой переменные как volatile объявлять не нужно.

JMM Monitor Lock Rule



```
public class MonitorLockRuleExample {

    private static class Box {
        private final Lock lock = new ReentrantLock(true);
        private long value = 0;

        public void increase(int diff) {
            lock.lock();
            try {
                this.value += diff;
            } finally {
                lock.unlock();
            }
        }
    }

    public static void main(String[] args) {
        Box box = new Box();
        var futures = Stream.generate(() -> CompletableFuture.runAsync(() -> box.increase(1)))
            .limit(10)
            .toArray(CompletableFuture[]::new);
        CompletableFuture.allOf(futures).join();
        System.out.println(box.value);
    }
}
```

А что с Atomic и примитивами?



Под капотом используют volatile и synchronized



Поэтому наследуют все те же правила

```
public class AtomicLong extends Number implements Serializable {  
  
    private static final long serialVersionUID = 1927816293512124184L;  
    static final boolean VM_SUPPORTS_LONG_CAS = VMSupportsCS8();  
    private static final Unsafe U = Unsafe.getUnsafe();  
    private static final long VALUE;  
    private volatile long value;  
  
    private static native boolean VMSupportsCS8();  
  
    public AtomicLong(long initialValue) { this.value = initialValue; }  
  
    public AtomicLong() {  
    }  
}
```

Singleton



Что не так с реализацией?

```
public class Singleton {  
    private static final Singleton instance = new Singleton();  
  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

Singleton

- ➡ Что не так с реализацией?
- ➡ А если все таки объект никогда не понадобится
- ➡ Нет ленивой инициализации

```
public class Singleton {  
    private static final Singleton instance = new Singleton();  
  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

Singleton



Что не так с реализацией?



```
public class Singleton {  
    private static Singleton instance;  
  
    public static Singleton getInstance( ) {  
        if (instance == null) {  
            instance = new Singleton( );  
        }  
        return instance;  
    }  
}
```

Singleton



Что не так с реализацией?



А как же потокобезопасность?



Нужен synchronized



```
public class Singleton {  
    private static Singleton instance;  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

Singleton



Что не так с реализацией?

```
public class Singleton {  
    private static Singleton instance;  
  
    public static synchronized Singleton getInstance( ) {  
        if (instance == null) {  
            instance = new Singleton( );  
        }  
        return instance;  
    }  
}
```

Singleton



Что не так с реализацией?



Каждый раз будем
синхронизироваться – это
дорого



Нужен double check locking

```
public class Singleton {  
    private static Singleton instance;  
  
    public static synchronized Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

Singleton



Что не так с реализацией?

```
public class Singleton {  
    private static Singleton instance;  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            synchronized (Singleton.class) {  
                if (instance == null) {  
                    instance = new Singleton();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

Singleton



Что не так с реализацией?



Теперь Java monitor rule не будет работать



Нам нужен volatile, чтобы синхронизировать кэши



```
public class Singleton {  
    private static Singleton instance;  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            synchronized (Singleton.class) {  
                if (instance == null) {  
                    instance = new Singleton();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

Singleton



Что не так с реализацией?

```
public class Singleton {  
    private static volatile Singleton instance;  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            synchronized (Singleton.class) {  
                if (instance == null) {  
                    instance = new Singleton();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

Singleton



Что не так с реализацией?



На самом деле все ок



Два чтения volatile

переменной – это тоже
дорого

```
public class Singleton {  
    private static volatile Singleton instance;  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            synchronized (Singleton.class) {  
                if (instance == null) {  
                    instance = new Singleton();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

Singleton



Теперь все будет ок

```
public class Singleton {  
    private static volatile Singleton instance;  
  
    public static Singleton getInstance() {  
        Singleton result = instance;  
        if (result != null) {  
            return result;  
        }  
        synchronized (Singleton.class) {  
            if (instance == null) {  
                instance = new Singleton();  
            }  
            return instance;  
        }  
    }  
}
```

Singleton Final



Но можно короче



До вызова getInstance
статический инициализатор
SingletonHolder не
выполнится



А HOLDER_INSTANCE –
финальное поле, поэтому
будет безопасная публикация

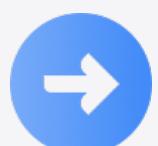
```
public class Singleton {  
  
    private static class SingletonHolder {  
        public static final Singleton HOLDER_INSTANCE = new Singleton();  
    }  
  
    public static Singleton getInstance() {  
        return SingletonHolder.HOLDER_INSTANCE;  
    }  
}
```



ТИНЬКОФФ

Java virtual threads

Virtual threads



А чем нас обычные то не
устраивают?



Threads



**Вспомним про обычные
потоки**

○

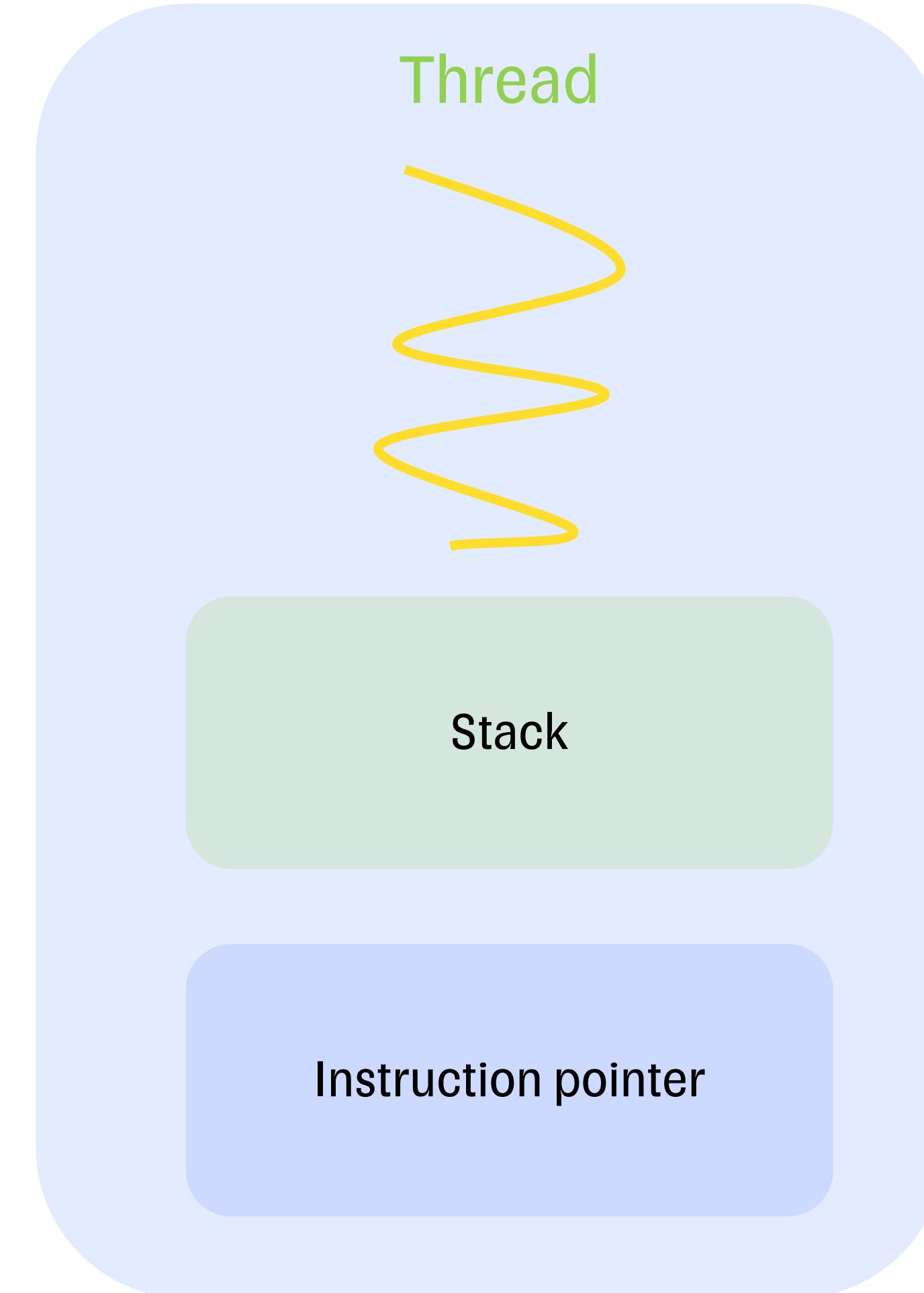
Threads



Вспомним про обычные потоки

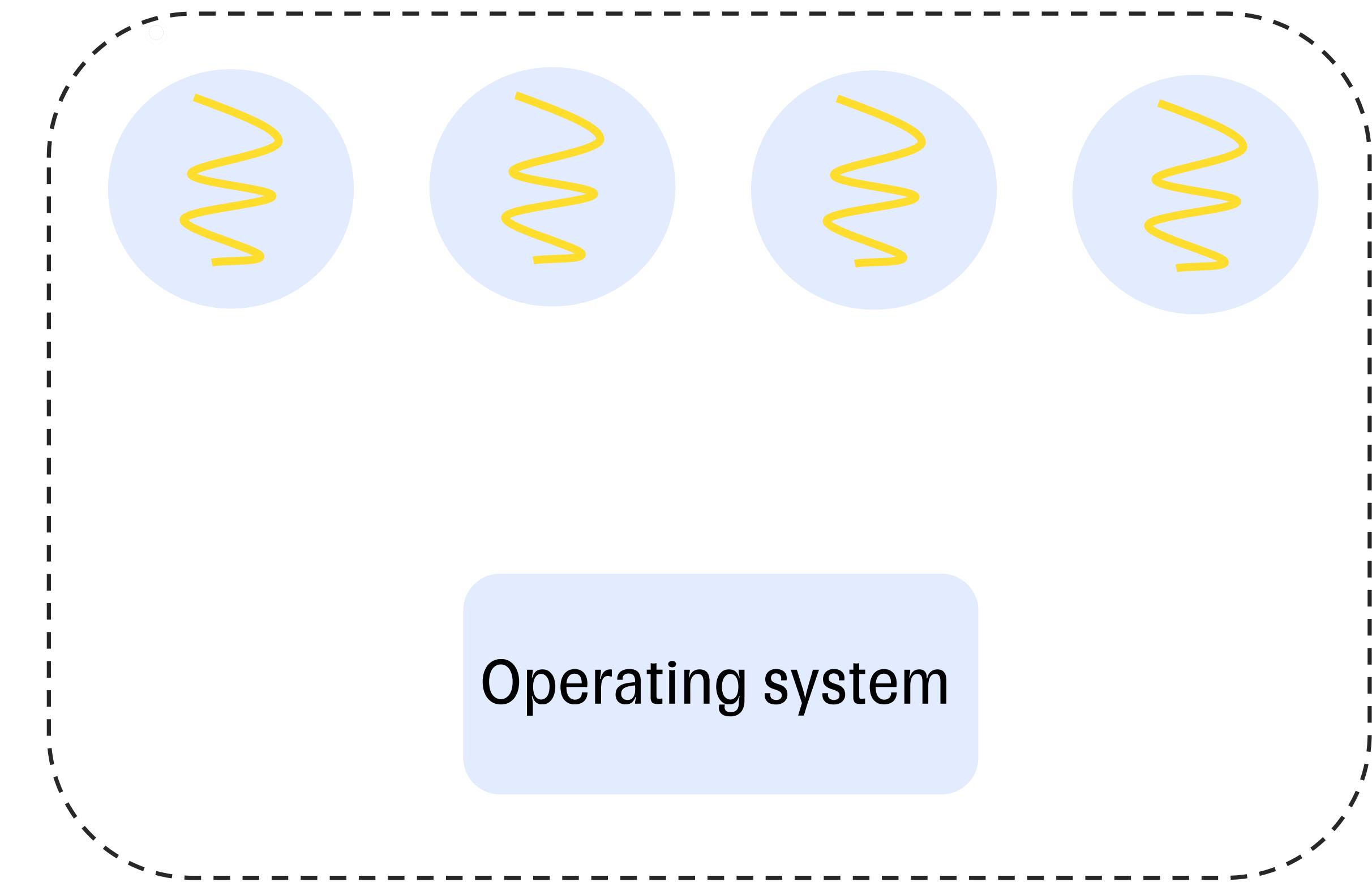


Stack – довольно увесистая вещь



Threads

- ➡ Вспомним про обычные потоки
- ➡ Stack – довольно увесистая вещь 2Mb
- ➡ Потоками управляет ОС
- ➡ Переключение контекстов



Пример



**Посмотрим сколько можно
создать потоков.**



**Посмотрим в консоль и
обнаружим ошибку и
посмотрим на цифру**

```
public class Main {  
  
    public static void main(String[] args) {  
        for (int i = 0; i < 1_000_000; i++) {  
            Thread.ofPlatform().start(() -> {  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    throw new RuntimeException(e);  
                }  
            });  
            System.out.println(i);  
        }  
    }  
}
```

Пример

→ Посмотрим сколько можно создать виртуальных потоков.

→ Посмотрим в консоль и обнаружим ошибку и посмотрим на цифру

```
public class Main {  
  
    public static void main(String[] args) {  
        for (int i = 0; i < 1_000_000; i++) {  
            Thread.ofVirtual().start(() -> {  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    throw new RuntimeException(e);  
                }  
            });  
            System.out.println(i);  
        }  
    }  
}
```

Пример

→ Посмотрим сколько можно создать виртуальных потоков.

→ Посмотрим в консоль и обнаружим ошибку и посмотрим на цифру

→ Создать можно много)

```
public class Main {  
  
    public static void main(String[] args) {  
        for (int i = 0; i < 1_000_000; i++) {  
            Thread.ofVirtual().start(() -> {  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    throw new RuntimeException(e);  
                }  
            });  
            System.out.println(i);  
        }  
    }  
}
```

Пример



Можно заменять на
виртуальные



Пример

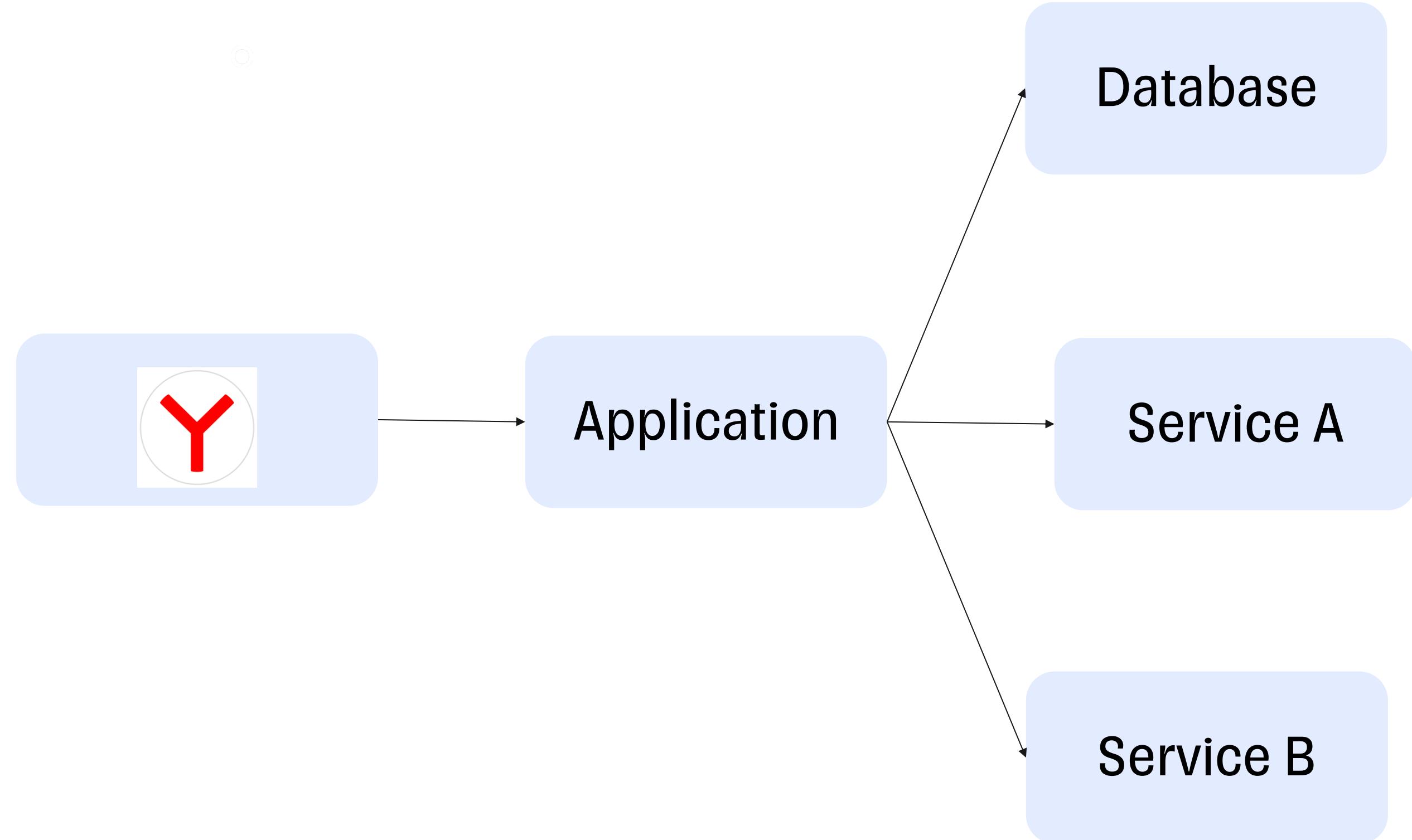
- Можно заменять на виртуальные
- Но все не так просто



Сервисы

Часто наше приложение
ходит по сети в другие
системы

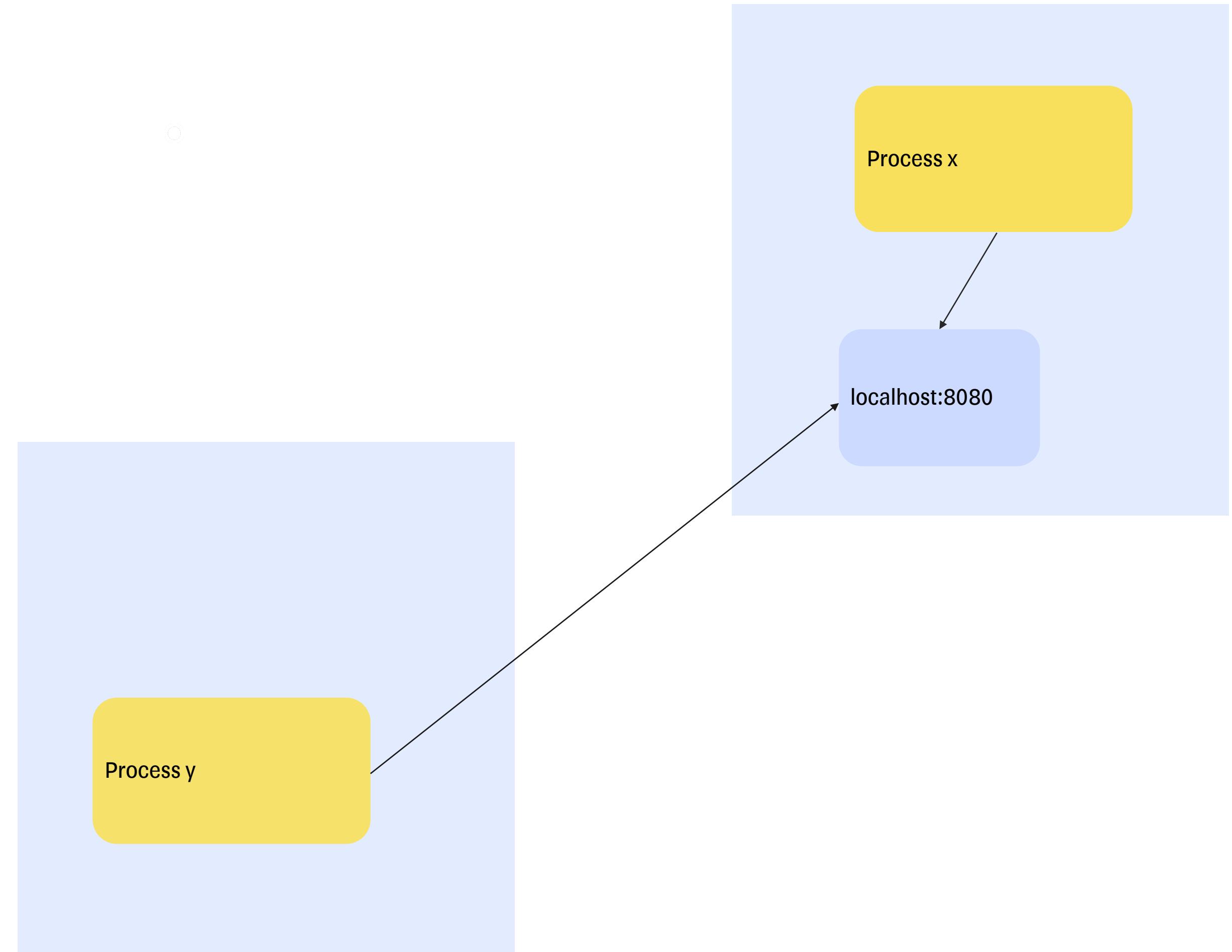
Когда мы ходим по сети мы
используем IO операции



Network



Вспоминаем семинар и
лекцию про сокеты



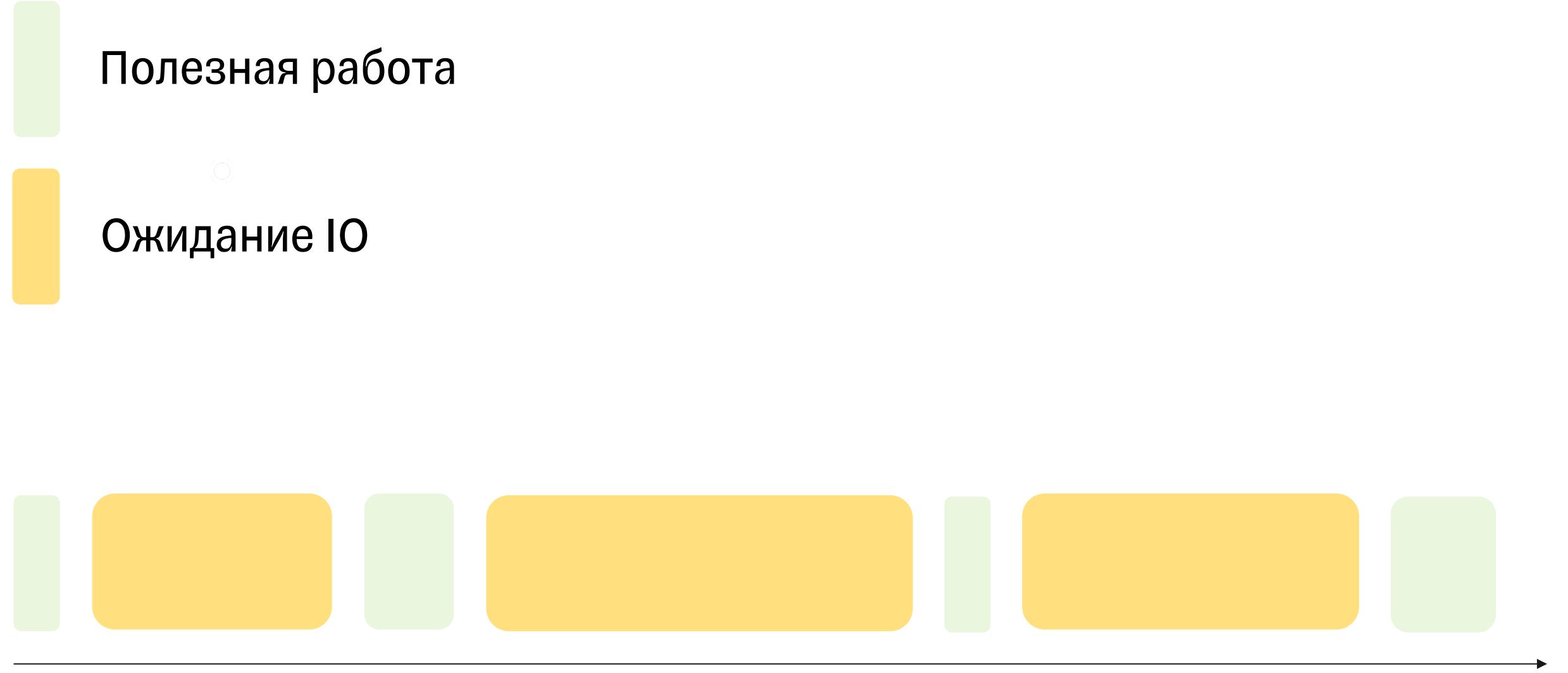
Обработка запроса



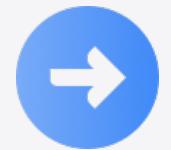
Ожидание IO - это походы в другие системы



Полезная работа – работа, которая требует расходов CPU



Обработка запроса



Ожидание IO - это походы в другие системы



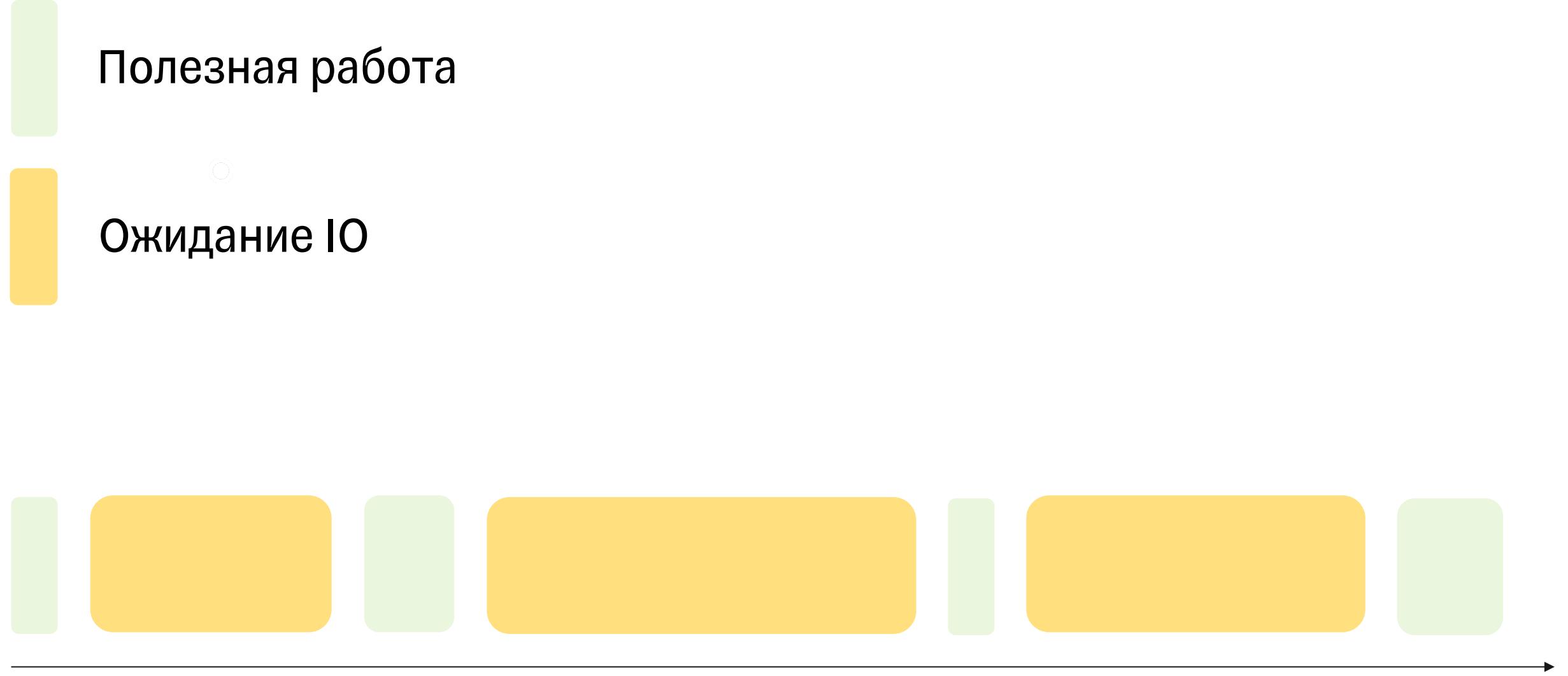
Полезная работа – работа, которая требует расходов CPU



Большую часть времени CPU используется в холостую



Для IO задач можно использовать виртуальные потоки



Использование в коде

Достаточно просто создать `virtualThreadPerTaskExecutor`



```
public class VirtualThreadExecutorExample {

    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        ExecutorService executorService = Executors.newVirtualThreadPerTaskExecutor();
        var futures = Stream.generate(() -> CompletableFuture.runAsync(
                VirtualThreadExecutorExample::downloadImage, executorService))
                .limit(10)
                .toArray(CompletableFuture[]::new);
        CompletableFuture.allOf(futures).join();
        executorService.shutdown();
        System.out.println("Finish download images " + (System.currentTimeMillis() - start));
    }

    private static void downloadImage() {
        try {
            URL url = new URL("https://source.unsplash.com/featured/300x201");
            InputStream in = new BufferedInputStream(url.openStream());
            Path path = Paths.get("image-" + UUID.randomUUID() + ".jpg");
            Files.write(path, in.readAllBytes());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Библиотеки



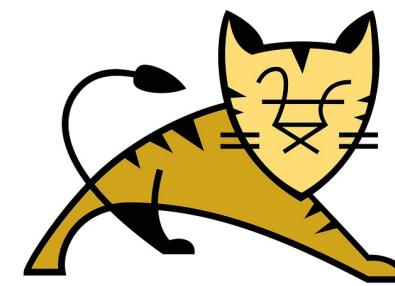
Можно заменить пулы на
виртуальные



При использовании
виртуальных тредов в tomcat
он смог обработать 10К
запросов



В дальнейшем библиотеки
получат такую возможность



Apache Tomcat

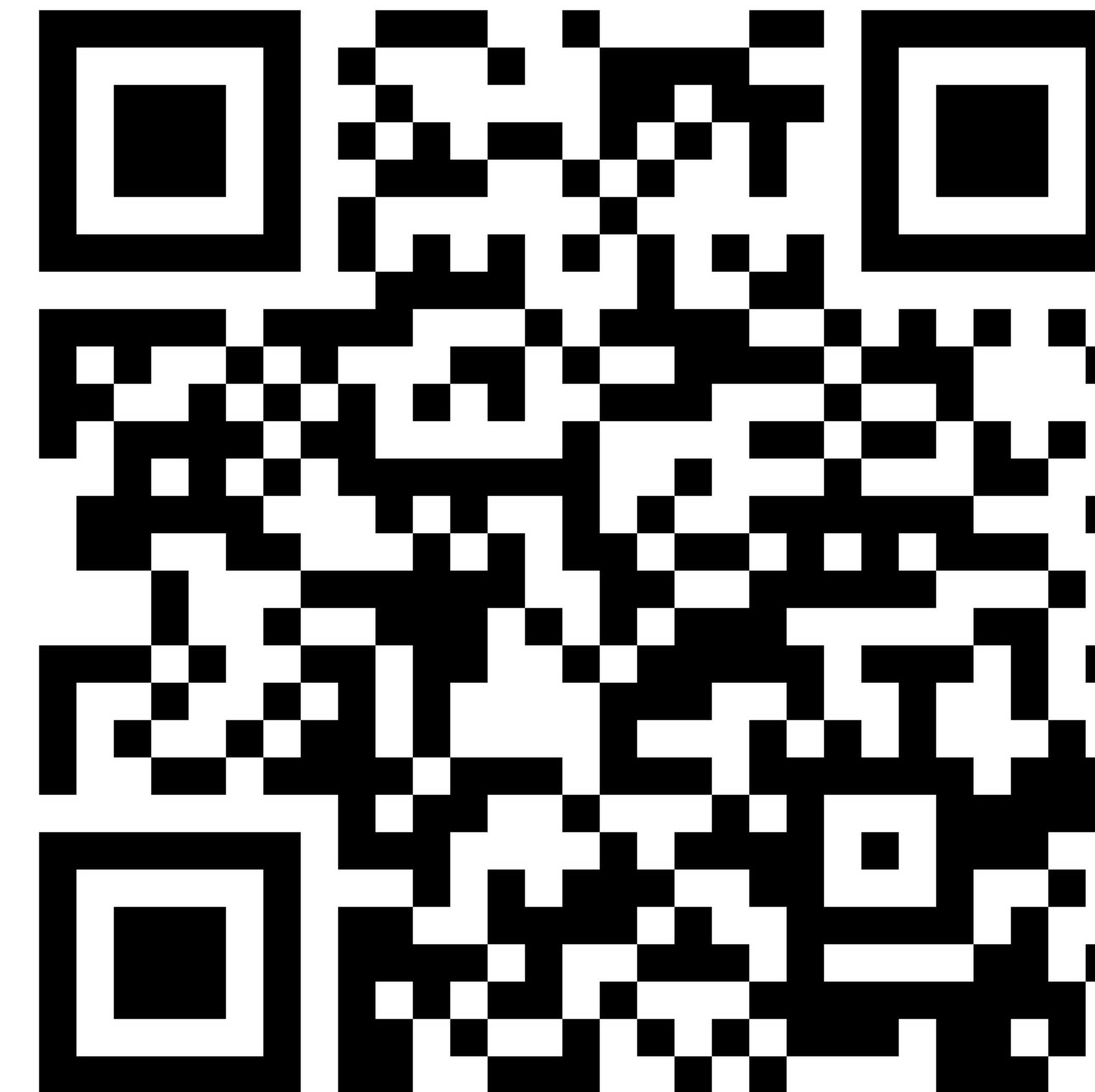


Доп. ресурсы

- <https://belief-driven-design.com/looking-at-java-21-virtual-threads-bd181/> - виртуальные треды
- <https://www.youtube.com/watch?v=iB2N8aqwtxc&t=1619s> – доклад А. Шепелева про JMM
- https://www.youtube.com/watch?v=qADk_tj4wY8&t=2249s – доклад с Jprime The JMM explained
- <https://www.youtube.com/watch?v=x1VZo-SPl1k&t=1151s> – доклад про Project Loom
- <https://habr.com/ru/articles/129494/> - статья про Singleton

Обратная связь

<https://polls.tinkoff.ru/s/clnd1dk6e001t01kfamzl3drs>





ТИНЬКОФФ

Он такой один

